7CS5A COMPILER CONSTRUCTION (Intermediate Code Generation)

Lecture 6: Oct. 18, 2018

Instructor: K.R. Chowdhary *: Professor of CS*

Disclaimer: *These notes have not been subjected to the usual scrutiny reserved for formal publications. They may be distributed outside this class only with the permission of the Instructor.*

6.1 Introduction

In the analysis-synthesis model of a compiler, the *front end* analyzes a source program and creates an intermediate representation, from this onwards the *back end* works and generates the target code. In ideal sense, details of the source language are confined to the front end, and details of the target machine[1] to the back end. With a properly defined intermediate representation, a compiler for any language i and any machine j can then be built by combining the front end for language i with the back end for machine j. This approach to creating suite of compilers can save a considerable amount of effort: $m \times n$ compilers can be built by writing just m front ends and n back ends. If $m = n$, then by writing just n front-ends, and n back-ends (total $2n$ code number of codes), we can construct $n \times n = n^2$ number of compilers, or by just 20 number of codes, it is possible to create 100 compilers – a very good saving in efforts for coding.

This chapter deals with intermediate representations, static type checking, and intermediate code generation. For simplicity, we assume that a compiler front end is organized as shown in Fig. 6.1, where parsing, *static checking*, and intermediate-code generation are done sequentially; sometimes they can be combined and folded into parsing. We shall use the syntax-directed formalisms in the chapters studied earlier to specify checking and translation. Many of the translation schemes can be implemented during either bottom-up or top-down parsing, using the techniques of syntax directed translation. All schemes can be implemented by creating a syntax tree and then walking the tree.

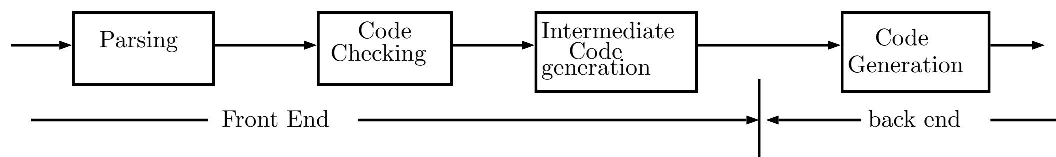

Figure 6.1: Logical structure of a compiler front end.

Static checking includes type checking, which ensures that operators are applied to compatible operands. It also includes any syntactic checks that remain after parsing. For example, static checking assures that a *break-statement* in *C* is enclosed within a *while-*, *for-*, or *switch-statement*; an error is reported if such an enclosing statement does not exist.

[1] In case of language like Fortran, C and C++, is machine hardware, and in case of Java it is JVM(java virtual machine – a software)

There can be a wide range of intermediate representations, including syntax trees and three-address code. The term "three-address code" comes from instructions of the form $x = y\ op\ x$ with three addresses: two for the operands y and x and one for the result x.

In the process of translating a program in a given source language into code for a given target machine, a compiler may construct a sequence of intermediate representations, as shown in Fig. 6.2. High-level representations are close to the source language and low-level representations are close to the target machine. Syntax trees are high level; they depict the natural hierarchical structure of the source program and are well suited to tasks like static type checking. It is suited for static checking because the hierarchical structure of a language is fixed in structure.

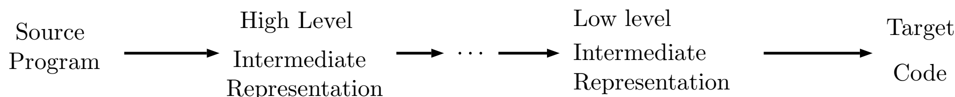

Figure 6.2: A compiler might use a sequence of intermediate representations.

A low-level representation is suitable for machine-dependent tasks like register allocation and instruction selection. Three-address code can range from high-level to low-level, depending on the choice of operators. For expressions, the differences between syntax trees and three-address code are superficial. For looping statements, for example, a syntax tree represents the components of a statement, whereas three-address code contains labels and jump instructions to represent the flow of control, as in machine language. For example the following sequence of statements are close to the assembly language.

$$\begin{aligned} label1: \; & x_i = y_i + z_i \\ & i = i + 1 \\ & if\ x < 100\ goto\ label1 \\ & next\ statement \end{aligned}$$

The choice or design of an intermediate representation varies from compiler to compiler. An intermediate representation may either be an actual language or it may consist of internal data structures that are shared by phases of the compiler. C is a programming language, yet it is often used as an intermediate form because it is flexible, it compiles into efficient machine code, and its compilers are widely available. The original C++ compiler consisted of a front end that generated C, treating a C compiler as a back end.

6.2 Applications of Syntax-Directed Translation

The syntax-directed translation techniques discussed here will be applied later in type checking and intermediate-code generation. The main application of SDD is in construction of syntax trees. Since some compilers use syntax trees as an intermediate representation, a common form of SDD turns its input string into a tree. To complete the translation to intermediate code, the compiler may then walk the syntax tree, using another set of rules that are in effect an SDD on the syntax tree rather than the parse tree. Later, we will discuss approaches to intermediate-code generation that apply an SDD without ever constructing a tree explicitly.

We consider two SDD's for constructing syntax trees for expressions. The first, an S-attributed definition, which is suitable for use during bottom-up parsing. The second, L-attributed, is suitable for use during top-down parsing. The final example of this section is an L-attributed definition that deals with basic and array types.

Construction of Syntax Trees. We know that each node in a syntax tree represents a construct; the children of the node represent the meaningful components of the construct. A syntax-tree node representing an expression "$E_1 + E_2$" has label + and two children representing the subexpressions E_1 and E_2.

We can implement the nodes of a syntax tree by objects with a suitable number of fields. Each object will have an *op* field that is the label of the node. The objects will have additional fields as follows:

- If the node is a leaf, an additional field holds the lexical value for the leaf. A constructor function $Leaf(op, val)$ creates a leaf object. Alternatively, if nodes are viewed as records, then $Leaf$ returns a pointer to a new record for a leaf.
- If the node is an interior node, there are as many additional fields as the node has children in the syntax tree. A constructor function $Node$ takes two or more arguments: $Node(op, c_1, c_2, ..., c_k)$ creates an object with first field *op* and k additional fields for the k children $c_1, ..., c_k$.

6.3 Variants of Syntax Trees

Nodes in a syntax tree represent constructs in the source program; the children of a node represent the meaningful components of a construct. A directed acyclic graph (hereafter called a DAG) for an expression identifies the common subexpressions (subexpressions that occur more than once) of the expression. As we will see that DAGs can be constructed by using the same techniques that construct syntax trees.

Like the syntax tree for an expression, a DAG has leaves corresponding to atomic operands and interior codes corresponding to operators. The difference is that a node N in a DAG has more than one parent if N represents a common subexpression; in a syntax tree, the tree for the common subexpression would be replicated as many times as the subexpression appears in the original expression. Thus, a DAG not only represents expressions more succinctly, it gives the compiler important clues regarding the generation of efficient code to evaluate the expressions.

Example 6.1 *Construct DAG for the expression:*

$$a + a * (b - c) + (b - c) * d.$$

Solution. Figure 6.3 shows the DAG for above expression. The leaf for a has two parents, because a appears twice in the expression. More interestingly, the two occurrences of the common subexpression $b - c$ are represented by one node, the node labeled $-$. That node has two parents, representing its two uses in the subexpressions $a * (b - c)$ and $(b - c) * d$. Even though b and c appear twice in the complete expression, their nodes each have one parent, since both uses are in the common subexpression $b - c$. □

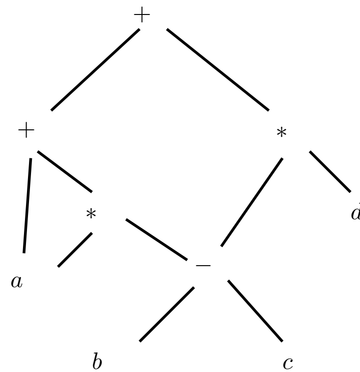

Figure 6.3: Dag for the expression $a + a * (b - c) + (b - c) * d$.

The SDD of Table 6.1 can be used to construct either syntax trees or a DAG's. The functions $Leaf$ and $Node$, create a fresh node each time they are called. It will construct a DAG if, before creating a new node, these functions first check whether an identical node already exists. If a previously created identical node exists, the existing node is returned. For instance, before constructing a new node, $Node(op, left, right)$ we check whether there is already a node with label *op*, and children $left$ and *right*, in that order. If so, the function *Node* returns the existing node; otherwise, it creates a new node.

Table 6.1: Syntax-directed definition to produce syntax trees or DAG's

S.No.	Production	Semantic Rules
1)	$E \rightarrow E_1 + T$	$E.node = new\ Node('+', E.node, T.node)$
2)	$E \rightarrow E_1 - T$	$E.node = new\ Node('-', E.node, T.node)$
3)	$E \rightarrow T$	$E.node = T.node$
4)	$T \rightarrow (E)$	$T.node = E.node$
5)	$T \rightarrow id$	$T.node = new\ Leaf(id, id.entry)$
6)	$T \rightarrow num$	$T.node = new\ Leaf(num, num.val)$

Example 6.2 *constructing the DAG of Fig. 6.3.*

Solution. The sequence of steps shown in Table 6.2 constructs the DAG of Fig. 6.3, provided $Node$ and $Leaf$ return an existing node, if possible. We assume that *entry-a* points to the symbol-table entry for a, and similarly for the other identifiers.

When the call to *Leaf(id, entry-a)* is repeated at step 2, the node created by the previous call is returned, so $p_2 = p_1$. Similarly, the nodes returned at steps 8 and 9 are the same as those returned at steps 3 and 4 (i.e., $p_8 = p_3$ and $p_8 = p_4$). Hence the node returned at step 10 must be the same at that returned at step 5; i.e., $p_{10} = p_9$. □

6.4 Exercises

1. Construct DAG for the expression:

$$((x + y) - ((x + y))) + ((x + y) * (x - y))$$

Table 6.2: Steps for constructing the DAG of Fig. 6.3

1)	$p_1 = Leaf(id, entry\text{-}a)$
2)	$p_2 = Leaf(id, entry\text{-}a) = p_1$
3)	$p_3 = Leaf(id, entry\text{-}b)$
4)	$p_4 = Leaf(id, entry\text{-}c)$
5)	$p_5 = Node('-', p_3, p_4)$
6)	$p_6 = Node('*', p_1, p_5)$
7)	$p_7 = Node('+', p_1, p_6)$
8)	$p_8 = Leaf(id, entry\text{-}b) = p_3$
9)	$p_9 = Leaf(id, entry\text{-}c) = p_4$
10)	$p_{10} = Node('-', p_3, p_4) = p_5$
11)	$p_{11} = Node(id, entry\text{-}d)$
12)	$p_{12} = Node('*', entry\text{-}d)$
13)	$p_{13} = Node('+', p_7, p_{12})$

2. Construct the DAG and identify the value numbers for the subexpressions of the following expressions, assuming associates from the left.
   (a) $a + b + (a + b)$.
   (b) $a + b + a + b$.
   (c) $a + a + ((a + a + a + (a + a + a + a))$.
3. For expressions, the differences between syntax trees and three-address code are superficial. Justify this statement.
4. Assume there are M number of total hardware (machines) available, and there are total L number of high level languages.
   (a) How many maximum number of compilers are required so that every language $l \in L$ can run on every machine $m \in M$? Assume that the front-end and back-end of compilers are not separately constructed.
   (b) Let there is concept of front-end and back-end parts of compilers, such that for many front-ends there can be single back-end and vice-versa. How many minimum number of compilers are required in the above case?
   (c) What are the possible advantages and disadvantages of the above configurations?
5. How the hierarchical structure of high level languages is helpful in static type-checking? Give examples.
6. Represent the following expressions using DAG.
   (a) $a * b + c * d$
   (b) $a * a * a$
   (c) $(b + c)^3$
   (d) $(a + b)^2(c + d)^3$